Architecture:

MVC (Model-View-Controller)

Divides an application into three main components:

* Model:
  + Manages the data and logic of the application
* View:
  + Responsible for the presentation and user interface
* Controller:
  + Handles the interaction between the Model and the View
* Promotes modularity and scalability of applications.

This structure helps us separate the different responsibilities of each part of our game. Additionally, dividing it in modules helps with maintenance and it makes it easier to collaborate between the team members. Finally, this structure helps prevent anti-patterns, such as Spaghetti Code, with a clear structure. We divide the code into these parts:

* **Model**: Manages the game data and logic, such as player status, scores, and interaction logic with environmental elements.
* **View**: Renders the game interface, such as the HUD (indicators for health, time, and comfort).
* **Controller**: Captures user input and translates it into commands for the model or view, such as interaction with driving controls and responses to AI events.

# Design Patterns Used:

**1. Singleton Pattern**

* **Principle**: The Singleton pattern ensures that a class has only **one instance** and provides a global point of access to that instance.
* **How It Works**: The class itself controls the instantiation process by storing its single instance and providing a static method to access it. This prevents other classes from creating new instances of the class.
* **Use Case**: Used for classes like GameManager and AudioManager to ensure that there is only one instance of these managers during the game's lifecycle.

**2. Observer Pattern**

* **Principle**: The Observer pattern defines a **one-to-many dependency** between objects, so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically.
* **How It Works**: The subject maintains a list of observers and provides methods to add, remove, and notify them of changes. When the subject’s state changes, it calls a method (e.g., notify()) to update all registered observers.
* **Use Case**: Ideal for implementing the comfort bar and notifications of changes in the game state, allowing the view to update information when the model changes.

**3. Factory Method Pattern**

* **Principle**: The Factory Method pattern provides an interface for **creating objects** but allows subclasses to alter the type of objects that will be created.
* **How It Works**: Instead of calling a constructor directly, a method (the factory method) is used to create an object. This approach promotes loose coupling by abstracting the creation process and delegating it to subclasses or dedicated factory classes.
* **Use Case**: Useful for dynamically creating instances of obstacles and interactive objects during the game.

# UML

* **GameManager (Singleton)**
  + **Methods**: StartGame(), EndGame(), RestartGame()
  + **Relation**: Manages the game cycle logic and coordinates between the PlayerController and AIController.
* **PlayerController (Controller)**
  + **Methods**: Move(), InteractWithEnvironment(), PickUpPassenger()
  + **Relation**: Controls player actions and sends notifications to the HUDController using the Observer pattern.
* **HUDController (View)**
  + **Methods**: UpdateScore(), DisplayTime(), UpdateComfortBar()
  + **Relation**: Subscribed to the GameManager and PlayerController to display changes on the HUD.
* **AIController (Model/Controller)**
  + **Methods**: ChasePlayer(), Patrol()
  + **Relation**: Implements the AI for police cars and controls interactions with the player.
* **ObstacleFactory (Factory Method)**
  + **Methods**: CreateObstacle(), CreatePowerUp()
  + **Relation**: Generates instances of obstacles and interactive objects, managed by the GameManager.
* **AudioManager (Singleton)**
  + **Methods**: PlaySound(), StopSound()
  + **Relation**: Manages game sound effects, subscribed to key events from the GameManager.